Lock-Free Hopscotch Hashing

Robert Kelly* Barak A. Pearlmutter† Phil Maguire‡

Abstract

In this paper we present a lock-free version of Hopscotch Hashing. Hopscotch Hashing is an open addressing algorithm originally proposed by Herlihy, Shavit, and Tzafrir [10], which is known for fast performance and excellent cache locality. The algorithm allows users of the table to skip or jump over irrelevant entries, allowing quick search, insertion, and removal of entries. Unlike traditional linear probing, Hopscotch Hashing is capable of operating under a high load factor, as probe counts remain small. Our lock-free version improves on both speed, cache locality, and progress guarantees of the original, being a chimera of two concurrent hash tables. We compare our data structure to various other lock-free and blocking hashing algorithms and show that its performance is in many cases superior to existing strategies. The proposed lock-free version overcomes some of the drawbacks associated with the original blocking version, leading to a substantial boost in scalability while maintaining attractive features like physical deletion or *probe-chain* compression.

1 Introduction

The trend in modern hardware development has shifted away from enhancing serial hardware performance towards multi-core processing. This trend forces programmers and algorithm designers to shift their thinking to a parallel mindset when writing code and designing their algorithms. Concurrent algorithms tackle the problem of sharing data and keeping that data coherent while multiple actors simultaneously attempt to change or access the data. For concurrent data structures and algorithms to perform well on modern processors they must generally have two properties. First, they must use the processor's cache memory efficiently for both data and instruction. Today's processors are very sensitive to memory access patterns and contention induced by concurrency in cache coherence protocols. As such, algorithm designers must take special care to accommodate these particulars. Second, the algorithms must ensure that reading the data structure is as cheap as possible. The majority of operations on structures like hash tables are read operations, meaning that it pays to optimise them for fast reads.

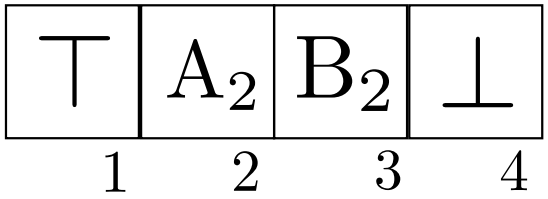


Figure 1: Table legend. The subscripts represent the optimal bucket index for an entry, the $\top$ represents a tombstone, and the $\bot$ represents an empty bucket. Bucket 1 contains the symbol for a tombstone, bucket 2 contains an entry A which belongs at bucket 2, bucket 3 contains another entry B which also belongs at bucket 2, and bucket 4 is empty.

Concurrent data structures and algorithms can be categorised into a variety of classes, with two prominent divisions emerging, namely *blocking* and *non-blocking*. Blocking algorithms generally involve the use of mutual exclusion primitives to give a single thread sole access to a specific location in memory. In contrast, non-blocking algorithms use low-level atomic primitives, such as *compare-and-swap*, to modify the data structure. The category of non-blocking algorithms contains a number of further subdivisions. Ordered by the strength of progress these are: obstruction-free [17] (individual progress in the case of no other contending operation), lock-free [18] (system progress but not individual thread or actor progress), wait-free [27] (every operation takes a finite number of steps to complete and must complete within that bound, which can depend on the number of threads or actors currently attempting such an operation), and wait-free population oblivious (every thread takes a finite amount of time regardless of how many other threads are in the system).

Maintaining strong progress conditions, or bounds on max latency, usually comes at the cost of throughput, meaning that the algorithms with the weakest guarantees typically boast the strongest real-world performance. Lock-free algorithms have the following benefits: freedom from deadlock, priority inversion, and

*Maynooth University Department of Computer Science. Email: rob.kelly@cs.nuim.ie

†Maynooth University Department of Computer Science. Email: barak@cs.nuim.ie

‡Maynooth University Department of Computer Science. Email: pmaguire@cs.nuim.ie

convoying. However, they suffer from their own set of challenges relating to memory management ([20], [19], [25], [24], [23]), proof of correctness ([7], [21], [22]) and poor performance under heavy write load, as excessive contention becomes an issue.

The outline of our paper is as follows. Section 2 gives the background for hash tables, a brief review of existing concurrent solutions, the original Hopscotch algorithm [10], the Purcell-Harris quadratic probing hash table [5], and the *K-CAS* [1] primitive. Section 3 outlines our algorithm with code and annotations. Section 4 details our proof sketch. Finally section 5 discusses the experimental performance results, contrasting them with those of competing algorithms.

2 Background

A data structure for which concurrency is particularly amenable is the hash table. A hash table is an efficient base structure for implementing the abstract data structure of sets or maps. In general, hash tables compute a *hash value* from a key or key-value pairing that a user seeks to either check membership, look up value, insert, or remove from the structure. The algorithm uses the hash value to index the location in which the entry should belong, and the entries are searched by following some strategy until a result is obtained. The expected lookup, insertion, and removal time bounds are $\mathcal{O}(1)$ [15]. Entries need only be capable of being hashed to a location and compared for equality. In contrast, tree structures require a total ordering on keys or key/value pairings, but don't require a hash function.

Hash-tables are bifurcated into either open addressing or closed addressing algorithms. Open addressing constrains a bucket to contain only a single key or key-value pair. This constraint means that if two different keys or key-value pairings hash to the same index for an insertion, then an algorithm must be devised for finding another suitable bucket for insertion. The algorithm must then also be able to find the key or key-value pairing at some bucket outside of the original/home index. The alternative approach is closed addressing. Closed addressing stores all keys or key-value pairs at the original hash index. If two keys or key-value pairs collide at an address, then they are stored in an auxiliary data-structure like a linked-list or binary tree for searching. Closed addressing is therefore relatively simple and concise, needing only to search a single bucket when examining the table for an entry. Open addressing can be more challenging, as buckets contain entries which don't belong there but rather are there due to a previous collision. There has been many publications covering both concurrent open addressing [2], [9], [5], [10], [12], [6], [11] and closed addressing algorithms [13], [3], [4].

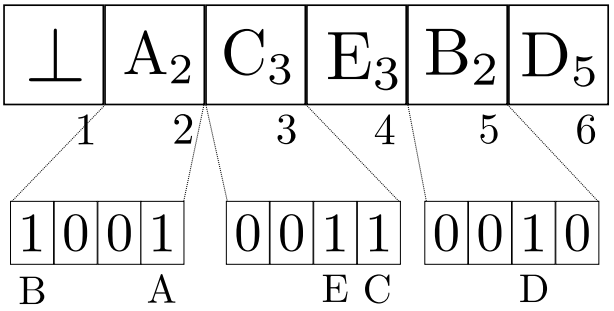
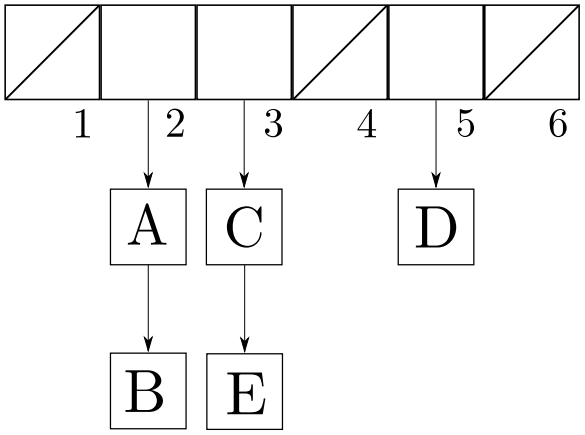


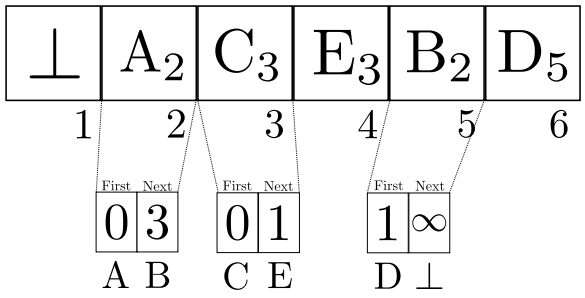
Figure 2: An example Hopscotch Hashing table. The neighbourhood of virtual buckets is represented by a bit-map below each bucket. Each set bit represents the offset of buckets to examine for key matches. Note the bit endianness of the bit-mask.

2.1 Original Hopscotch Hashing Herlihy, Shavit, and Tzafrir [10] presented Hopscotch Hashing, a hash table algorithm they describe as a mixture of linear probing [28], separate chaining [14], and Cuckoo Hashing [29]. Their paper presented solutions in both the serial and concurrent form. The algorithm comes in two main flavours. The first is to create a fixed sized *neighbourhood* defining a virtual bucket containing many physical buckets. This neighbourhood is represented with a bit-mask, where the index of each set bit in the mask indicates the presence of a relevant entry for that particular bucket. An example table is shown in Figure **2**, and a table legend is shown in Figure **1**. The algorithm solves collisions by linear probing to a free bucket and marking the i'th bit in the bit-mask, where i is the distance from the original bucket. Due to the fixed size of the virtual buckets, a limit is enforced on how far an entry can be placed from its home/original bucket. The authors describe an algorithm for displacing entries from the saturated neighbourhood in order to create space for the new entry. The displacement algorithm works by linear probing to find an initial bucket and marking it as `Busy`. The algorithm then subsequently relocates the bucket backwards, swapping it with a currently occupied bucket, and modifying the occupied bucket's bit-mask during the move. Moving an occupied bucket forwards is only permissible if the destination bucket is also inside its neighbourhood. The algorithm repeats this process until the initially claimed bucket is within range of its home/original neighbourhood. If no such displacement can be made, then the table is forced to resize.

The authors then build upon this idea of the fixed size neighbourhood, using *relative offsets* to indicate where the next relevant entry is stored. These offsets represent the hops throughout the table. Each entry



(a) A Separate Chaining table. Entries hashed to a bucket are put into a linked list at the bucket. Buckets with nulls are denoted with a diagonal line.



(b) A "relative offset" variant of Hopscotch Hashing table with the same entries as the Separate Chaining table. Each bucket contains two integers. The first is the offset where the probe chain starts, and the second is the next item in the probe chain.

Figure 3: Comparison between Separate Chaining and Hopscotch Hashing with relative offsets.

is then part of a chain of entries, aptly named the *probe chain*. The relative offsets, if large enough, can represent a neighbourhood as large as the table, removing the need to displace entries that otherwise would be outside the neighbourhood range. These hops, like the bit-masks, allow the method to skip over entries that are irrelevant to the search. For example, when a table using linear probing becomes saturated an entry may end up quite some distance from its original bucket. If such a situation were to arise in Hopscotch Hashing, then the last relevant entry to that original bucket would have the offset embedded into it, "pointing" to the new entry. The relative offset variant of Hopscotch can be thought of as a specialised version of Separate Chaining, in which the linked list present at each bucket has been flattened directly into the table. Figure **3** illustrates a comparison between Separate Chaining and Hopscotch Hashing where each table has the same entries.

The use of relative offsets does not mean that the need to relocate entries disappears. The authors (in their released implementation) optimise the probe chain by shifting entries backwards when an entry earlier in the chain is removed. Resizes may still be required, as some entries may end up being further away from the last item in the probe chain than can be represented in the relative offsets. We choose the fixed size bit-mask as our model for our lock-free version of the algorithm. Their concurrent version employs mutual exclusion on threads wishing to insert or remove from the data structure, with remove operations incrementing a relocation counter relating to the relevant bucket deleted or moved. The reading thread will check the relocation counter before and after, to ensure that none of the entries have been shifted around during the reading. The number of segments is set to the expected concurrency exposure of the table.

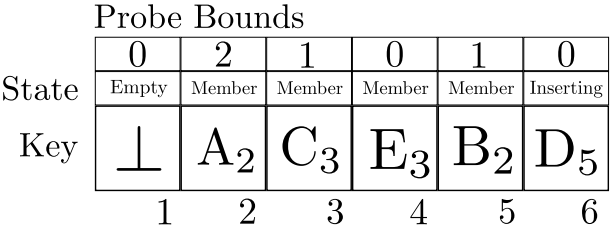


Figure 4: An illustration of a Purcell-Harris table with bucket states and probe bounds.

2.2 Purcell-Harris Algorithm Our algorithm uses the Purcell-Harris method for insertion and deletion, so as to support physical deletion. Their approach uses a state based method for insertion and deletion. During insertion, keys or key-value pairs are eagerly inserted into the table and later checked for uniqueness. A bucket in their algorithm can be in 1 of 6 states, namely `Empty`, `Busy`, `Collided`, `Visible`, `Inserting`, or `Member`. `Empty` indicates that this bucket is empty and available for use in a new insertion. `Busy` can be seen as a *lock*, used when the inserting/deleting thread is busy writing/deleting the key information in that bucket, and no one else may use the bucket. `Collided` is a state to indicate that the eager insertion of the

entry has failed, as either a closer *potential* entry exists, or an entry already marked as `Member` exists. `Visible` represents buckets which contain valid key data, but the bounds for per bucket probes may have not been updated yet. This allows other threads to see the entry and not decrease the probe length. The `Inserting` state means that the per bucket bounds have been updated to include this bucket and is in the process of being checked for uniqueness in the table. `Member` is the final stage of insertion, representing a unique key or key-value pair which is part of the table. All state variables contain an associated relocation counter to avoid the *ABA* problem [16]. An illustration of the table can be viewed in Figure **4**.

Removal of keys or key-value pairs is trivial, since they are atomically manipulated through the state variable. When removing an entry, the algorithm will simply move the state from `Member` to `Busy`, erase the key, potentially move the bound downwards, and then move the state back to `Empty`. Supporting physical deletion in non-blocking algorithms is difficult and is normally accomplished by putting each entry behind a dynamically allocated node. The Purcell-Harris algorithm makes this process trivial, while delivering good performance; all state variables and entries can be stored directly in the table, removing a level of indirection, and increasing cache efficiency. Checking for key membership is also straightforward. This simply involves reading the probe bound, examining any buckets marked as `Member` in the table, and checking that the associated version number hasn't changed since reading the state variable.

2.3 K-CAS *K-CAS* or, *multi-word-compare-and-swap*, is an extension of the *compare-and-swap* or, *CAS* primitive. The algorithm allows for multiple memory locations to be atomically updated in the same fashion as a single *CAS* operation. The quintessential algorithm for *K-CAS* was published by Harris, Kaiser, and Pratt [1], which works by installing shared operation *descriptors* at each word being updated so that threads can cooperatively help each other complete the operation. To distinguish descriptors from normal words, up to 2 bits are reserved by the algorithm. For pointers, no extra bits are needed, as the lower bits are normally free; normal values like integers require 2 bits to be set aside. As a result of reserving bits, special read/write functions are needed when interfacing with memory. The read function checks the reserved bits of a word to see if any ongoing *K-CAS* operation is currently in flight and, if so, assists in completing it.

The performance of *K-CAS* was previously limited due to the necessity of a memory reclaimer. Each descriptor must be *fresh* (newly allocated) to avoid the ABA problem [16], and, as such, the overhead was high. An algorithm by Arbel-Raviv and Brown [26] employs descriptor reuse, thereby eliminating the need for a freshly allocated descriptor for each operation. This substantially increases the performance of each *K-CAS* operation, making their new algorithm a practical consideration when designing performant lock-free algorithms.

3 Algorithm

In the following section we provide an overview of our lock-free version of Hopscotch Hashing. The blueprint of our data-structure is that of a concurrent set, conforming to the API and abstract semantics. Our algorithm starts with the Purcell and Harris implementation of lock-free quadratic probing, and uses Hopscotch's bit-masks to create a fix-bound probe range for the searches. The check for uniqueness is then performed within that fix-sized area. The combination of the two algorithms removes the need for conditionally raising or lowering probe bounds, and allows for Hopscotch searching, insertion, and deletion in a lock-free manner. Like Purcell and Harris' quadratic probing, it allows for physical deletion, a difficult task to perform for a lock-free algorithm. We employ relocation counters at each bucket to indicate when that bucket's neighbourhood has experienced a bucket relocation, a necessity seeing as our algorithm moves entries around the table. All operations read the relocation counter before and after to ensure that no concurrent move operations have taken place, thus ensuring operation consistency. Our algorithm also makes use of *multi-word-compare-and-swap* or *K-CAS* [1] for an atomic swap. Previous work by Kelly, Pearlmutter, and Maguire [11] shows that an efficient algorithm for *K-CAS* by Arbel-Raviv and Brown [26] is feasible in the construction of concurrent hash-tables.

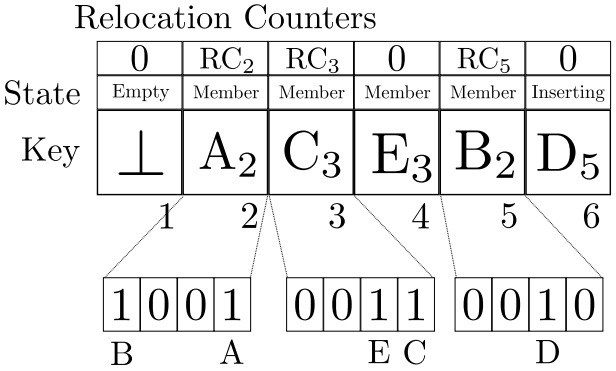


Figure 5: An example table for lock-free Hopscotch Insertion. Our algorithm blends the Purcell-Harris state based buckets with Hopscotch bit-map neighbourhoods, fixing the probe bounds.

When describing the table algorithm we start with the *Add* method, as it influences the design of all other methods. The insertion process begins by checking if the key is already present in the table, reading the neighbourhood mask, and checking relevant buckets as indicated by each set bit. This check, however, is optional, as entries are inserted eagerly and checked for uniqueness afterwards. The algorithm then linear probes to find an `Empty` bucket and claims it, marking it as `Busy`. If the bucket claimed is within the neighbourhood range of the bit-mask, then a uniqueness check begins, completing the insertion. Otherwise the claimed bucket must be moved backwards towards its home neighbourhood bit-mask. To move a bucket back we use the standard Hopscotch displacement method, finding a suitable bucket to move forward, copying its key or key-value pair to the new bucket, marking it as *live* in the neighbourhood bit-mask, and finally use *K-CAS* to swap the bucket states and increment the moved bucket's relocation counter to force re-reads of the neighbourhood. If the *K-CAS* is successful, we remove the bit previously corresponding to where the now moved entry previously resided. All of these searches also take account of ongoing relocations by reading relocation counters of buckets to ensure they don't miss a potential bucket. An illustration of the table is in Figure **5**.

Figure **6** gives a general overview of the insertion process, showing the many different stages for inserting an entry. Initially, claiming a bucket can be seen in Figure **6a**. This bucket is outside the neighbourhood range and needs to be relocated backwards. The algorithm scans for potential buckets, as seen in Figure **6b**, for a suitable entry to move. The first entry seen is entry `E`, which cannot be moved, as it would be outside its neighbourhood range. The next entry considered is `D`; moving it is legal. As the next entry, `B`'s location is marked in the bit-map for `D`, in anticipation of its movement. Following that, *K-CAS* is employed to swap the two entries and update `D`'s relocation counter. The resulting configuration can be seen in Figure **6c**, where `D` and `B` have been swapped and `D`'s relocation counter has been updated. The process continues as seen in Figure **6d**. `B` removes itself from `D`'s bit-mask, and then checks if `B` is within its neighbourhood range, which it is. Lastly, the algorithm marks the bit in `B`'s home neighbourhood bit-mask, and, after adding it to the neighbourhood, the Purcell-Harris uniqueness check is performed within that neighbourhood, making entry `B` a member of the table.

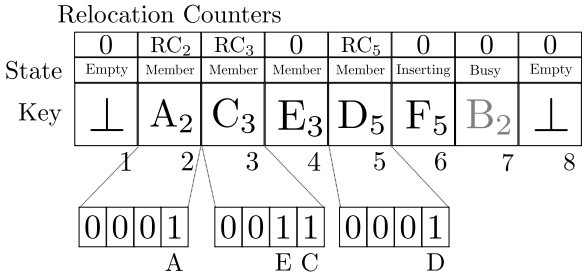
Contains and *Remove* remain relatively simple, both loading the relocation counters of the neighbourhood being examined, along with the bit-mask. *Contains* simply checks all buckets indicated by the bit-mask and successfully finishes if a key or key-value is found. If an entry isn't found, yet the relocation counter has changed, the method is performed again, returning an unsuccessful result if no change in the relocation counter is seen. *Remove* follows the same process as *Contains*, except once an matching entry is found it is put into a `Busy` state via a *CAS*. If successfully put into a `Busy` state, then the key is removed, the relevant bit unset in the neighbourhood bit-mask, and the bucket marked as `Empty` for reuse. *Remove* can also optionally compress probe chains by moving entries further away closer to the original bucket, optimising cache usage. A basic code walk-through follows, which outlines the most important parts of the algorithm.

A - Contains

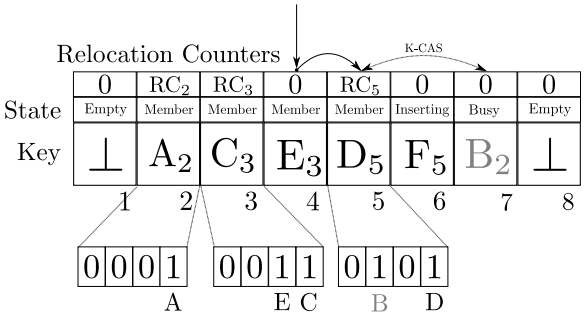
All line references relate to the code in Figure **7** The *Contains* method is relatively unchanged from the blocking version. *Contains* need not worry about interleaving *FindCloserBucket* calls interfering with its correctness, as a relocation counter check simply restarts the method if a change is detected. *Contains* calculates the initial starting bucket and loads the current relocation counter on lines **2 - 4**. Next it loads the current bit-mask for the original bucket, examining all bits set and calculating all indices to examine for key membership (lines **7 - 11**). Lines **13 - 16** load the state of the bucket, check whether the bucket is a `Member`, and subsequently proceed to check the key for a match. Following an unsuccessful search, lines **23 - 28** reload the relocation counter to check for a change, returning `false` if there hasn't been one, and running again if there was.

B - Add

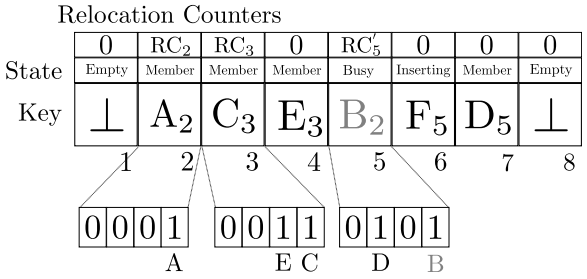
All of the following code lines refer to Figure **8**. *Add* can be broken down into four sections, one optional and the other three necessary. The first and optional section is to check if the key already exists in the table. The second section involves claiming an `Empty` bucket, the third involves moving that bucket to within neighbourhood range if necessary, while the fourth performs an exclusiveness check once the bucket is within range. Dealing with the first section, *Add* performs a general preamble for hash tables on lines **2 - 4**, calculating hash values and loading relocation counters. On line **6** *Add* can run an optional check of the table, to ensure the item being inserted is not already in the table. The code is more or less identical to that of *Contains* so we leave it out here. Next, in the second section (lines **8 - 17**), the algorithm attempts to reserve an `Empty` bucket as `Busy` up to some defined



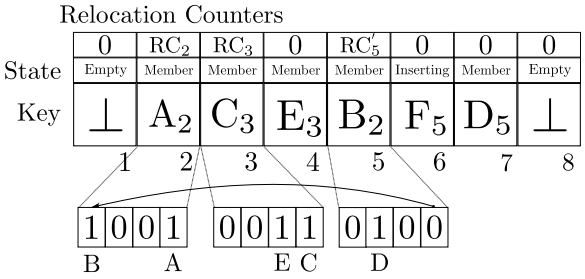
(a) Bucket **7** is claimed and marked as `Busy`. The bucket is outside the neighbourhood range and must be moved back.



(b) Find a closer bucket and swap it with bucket 7, adding the bucket to the bit-mask and incrementing the relocation counter when swapping.



(c) Updated table after swapping buckets 5 and 7. The relocation counter at bucket 5 has been changed.



(d) Confirm bucket 5's uniqueness in the table, making the state of bucket 5 `Member` and adding it to bucket 2's bit-map.

Figure 6: States of lock-free Hopscotch inserting element `B`.

probe limit. This limit is a user provided parameter. It represents the probe distance tolerated before a table resize is necessary. The same limit exists in the original blocking Hopscotch Hashing.

The third section checks whether the user distance has been violated, causing a resize if so (lines **22** and **36 - 37**. If the bucket claimed is within the `MAX_DISTANCE`, the algorithm checks if the distance is within neighbourhood range on line **26**, moving the claimed bucket backwards until it is in range. The algorithm moves the bucket closer by calling *FindCloserBucket* on line **27**, updating the reserved bucket and its offset every iteration until the bucket is within range. If the method *FindCloserBucket* returns and no progress has been made, the table is considered saturated and a call to *resize* is made on line **30**. Once it is within neighbourhood range, the bucket has its key written and state updated on lines **34 - 35**. The fourth and final section is a slightly modified Purcell-Harris uniqueness check, which instead searches within a fixed bound probe, looping on a relocation counter in case of concurrent relocation. The last modifications are to check the relocation counter before and after the exit point in the original Purcell-Harris method. The retry mechanism is identical to the likes of *Contains* and *Remove*.

C - Remove

All of the following code lines refer to Figure **9**. *Remove* is near identical to *Contains*, except that when a key match happens, the method tries to *CAS* the state variable from `Member` to `Busy` (lines **17 - 18**). Line **21** is where one could optionally compress the other entries backward, closer to their original bucket. Lines **24 - 26** remove the key from the table, remove the bit from the bit-mask, and set the bucket back to `Empty` for reuse. The algorithm for the preamble on lines **2 - 4** and relocation counters on lines **35 - 40** is the same as seen in *Contains*.

D - Find Closer Bucket

All of the following code lines refer to Figure **10**. The goal of *FindCloserBucket* (*FCB*) is to move back some bucket marked as `Busy` with another bucket which is already a member of the table. The use-case is when the `Add` method claims a bucket inside the `MAX_DISTANCE`, but outside the `NEIGHBOURHOOD_DISTANCE`. In this case, displacing a bucket already in the table is necessary. Like other methods, *FCB* loops while there is a relocation counter discrepancy, since another *FCB* has run and potentially

```
1  fn Contains(key: K) -> bool {
2    key_hash = hash(key);
3    ob = key_hash % size;
4    rc_before = table[ob].rc;
5    while(true) {
6      // Load the neighbourhood bit-mask
7      bm = table[ob].bm;
8      while(bm != 0) {
9        // Find lowest set bit
10       lsb = lowest_set_bit(bm);
11       index = ob + lsb;
12       // Purcell Harris bucket check.
13       _, state = table[index].vs;
14       if(state == Member) {
15         if(table[index].k == key) {
16           return true;
17         }
18       }
19       // Remove the bit just checked.
20       bm = XOR(bm, 1 << lsb);
21     }
22     // Check the relocation counter
23     rc_after = table[ob].rc;
24     if(rc_before == rc_after) {
25       return false;
26     }
27     // Check bit-mask again
28     rc_before = rc_after;
29   }
30 }
```

Figure 7: Pseudo-code for *Contains*

disrupted the result. Lines **6 - 7** calculate the maximum distance a bucket can be moved, and create a new *K-CAS* descriptor. The loop on line **8** examines each bucket in an earlier position than the one being relocated. The loop on line **11** examines the current bucket's bit-mask for a candidate to swap, and checks if that move is worthwhile. Swapping is conditional on two criteria. The first is that moving the candidate entry keeps it within its neighbourhood, the second is that the entry being swapped actually moves closer to its home neighbourhood. The first criterion is ensured by definition, as the loop's variables are initialised to all be legal swaps (lines **6 - 8**). The second criterion is checked on line **15** to ensure the swap actually moves the bucket closer to home.

Once a candidate bucket has been identified, its bit-mask is marked to include the destination bucket (line **21**). Lines **23 - 28** fill out the descriptor with all the information needed to swap the two buckets and increment the relocation counter. The *K-CAS* function is invoked on line **27**, either atomically swapping the two buckets, or else failing. Failing to swap the buckets results in the candidate's bit-mask having the destination bit unmarked, and the method is restarted. If the call

```
1  fn Add(key: K) -> bool {
2    key_hash = hash(key);
3    ob = key_hash % size;
4    rc_before = table[ob].rc;
5    // Part 1: Run an optional read of the table...
6    ...
7    // Part 2: Reserve a bucket
8    rb = ob;
9    offset = 0;
10   for(; offset < MAX_DISTANCE; rb++, offset++) {
11   retry:
12     v, s = table[rb].vs;
13     if(s == Empty) {
14       if(CAS(&table[rb].vs,
15         {v, Empty}, {v + 1 , Busy})) {
16         break;
17       } else { goto retry; }
18     }
19   }
20   // Part 3: Is the reserved bucket within -
21   // general range?
22   if(offset < MAX_DISTANCE) {
23     // Is the reserved bucket within -
24     // neighbourhood range?
25     before_rb = rb;
26     while(offset >= NEIGHBOURHOOD_DISTANCE) {
27         rb, offset = FindCloserBucket(rb, offset);
28         // No closer bucket found, resize.
29         if(rc == before_rb) {
30           resize();
31           ...
32         }
33     }
34     table[rb].k = key;
35     table[rb].vs = {v, Inserting};
36   } else {
37     // We need to resize the table.
38     resize();
39     ...
40   }
41   // Part 4: Modified Purcell-Harris
42   // exclusivity check.
43   ...
44 }
```

Figure 8: Pseudo-code for *Add*

succeeds, then the bit where the candidate used to be is unmarked from its bit-mask and the new bucket is returned on lines **35** and **36**. Lines **42 - 44** check for a relocation counter discrepancy, restarting the method if one is detected.

4 Proof Of Correctness and Progress

4.1 Correctness We present a simple sketch proof of correctness using lemmas to build up our proof argument. Each method will be evaluated for *linearisability* [7]. If every method is *linearisable*, then the entire object is *linearisable*. We deal with every code point in

```
1  fn Remove(key: K) -> bool {
2    key_hash = hash(key);
3    ob = key_hash % size;
4    rc_before = table[ob].rc;
5    while(true) {
6      // Load the neighbourhood bit-mask
7      bm = table[ob].bm;
8      while(bm != 0) {
9        // Find lowest set bit
10       lsb = lowest_set_bit(bm);
11       index = ob + lsb;
12     retry:
13       // Purcell Harris bucket check.
14       _, state = table[index].vs;
15       if(state == Member) {
16         if(table[index].k == key) {
17           if(CAS(&table[index].vs,
18                  {v, Member}, {v, Busy})) {
19             // Optionally: shift entries to -
20             // closer bucket
21             ...
22             // Remove key and the bit, -
23             // mark bucket as Empty
24             table[index].key = Nil;
25             table[ob].bm.fetch_xor(1 << lsb);
26             table[index].vs = {v + 1, Empty};
27             return true;
28           } else { goto retry; }
29         }
30       }
31       // Remove the bit just checked.
32       bm = XOR(bm, 1 << lsb);
33     }
34     // Check the relocation counter
35     rc_after = table[ob].rc;
36     if(rc_before == rc_after) {
37       return false;
38     }
39     // Check bit-mask again
40     rc_before = rc_after;
41   }
42 }
```

Figure 9: Pseudo-code for *Remove*

every method, highlighting the particular *linearisation* point in the algorithm and in the code.

The sketch of our proof argument is as follows. Both *Contains* and *Removes* read the relocation counters, then the bit-mask neighbourhood, and perform a combination of Hopscotch and Purcell-Harris state-based reads. After an operation is performed, the relocation counter is re-read, and the operation is performed again if a relocation is detected. Both methods can therefore be considered in isolation, as any abnormalities caused by moving entries around the table are dealt with by the relocation counters. A *linearisable Add* method must not fail to insert a key where there isn't one, and must not succeed in inserting a key where there already is one.

```
1  fn FindCloserBucket(rb: u64, offset: 64) -> {u64, u64} {
2    rv, rs = table[rb].vs;
3    while(true) {
4    begin:
5      // Move back as far as possible
6      dist = NEIGHBOURHOOD_DISTANCE - 1;
7      desc = create_descriptor();
8      for(cb = rb - dist; cb < rb; cb++, dist--) {
9        rc_before = table[cb].rc;
10       bm = table[cb].bm;
11       while(bm != 0) {
12         lsb = lowest_set_bit(bm);
13         i = ob + lsb;
14         // Check bucket only if advantageous to move
15         if(i >= rb) { break; }
16         iv, is = table[i].vs;
17         // Is this bucket a candidate?
18         if(is == Member) {
19           table[rb].k = table[i].k;
20           // Mark our bucket as active
21           table[cb].bm.fetch_or(1 << dist);
22           // Prepare the K-CAS descriptor
23           desc.add(&table[cb].rc, rc_before,
24             rc_before + 1);
25           desc.add(&table[i].vs, {iv, is},
26             {iv, Busy});
27           desc.add(&table[rb].vs, {rv, rs},
28             {rv, Member});
29           if(!K_CAS(desc)) {
30             // Turn off our bit preemptively turned on
31             table[cb].bm.fetch_xor(1 << dist);
32             goto begin;
33           }
34           // Unmark the now moved bucket, continue on.
35           table[cb].bm.fetch_xor(1 << lsb);
36           return { i, offset - dist };
37         }
38         // Remove the bit just checked.
39         bm = XOR(bm, 1 << lsb);
40       }
41       // Check the relocation counter
42       rc_after = table[ob].rc;
43       if(rc_before != rc_after) {
44         goto begin;
45       }
46     }
47     // Return the same bucket and offset to indicate failure.
48     return { rb, offset };
49   }
50 }
```

Figure 10: Pseudo-code for *FindCloserBucket*

The only part which makes a key a member of the table is the uniqueness check. This last component of *Add*, the uniqueness check, has to deal with concurrent *Add* calls moving entries around. It is the only component capable of spuriously making a key a `Member`, or deleting it. The first section is a linear probe to claim an `Empty` bucket as `Busy`, claiming it via a *CAS*. Once a bucket is

in the `Busy` state, it can only transition to another state by the thread that marked it as `Busy`. In other words, the thread has pseudo ownership of the bucket. The linear probing algorithm is simple and doesn't create any difficulty in reasoning about the concurrent correctness of the algorithm. All that matters is that a bucket is moved from `Empty` to `Busy`. The second stage is to move that claimed bucket to within neighbourhood range if not already inside. Moving is achieved by linearly probing towards the claimed bucket and atomically swapping it with a valid bucket found along the way. The probing for another bucket is performed from the max distance the bucket could move, that is, from a "neighbourhood distance" away. The atomic swap is accomplished by *K-CAS* so that swapping has no visible intermediate state and can retry if the operation fails. Swapping increments the relocation counters for the home bucket, forcing both *Contains* and *Remove* to re-run if necessary. Once the entry is moved within neighbourhood range, a uniqueness check is performed. The check is near identical to that found in the Purcell-Harris table, the only difference being a single extra step. The method must check the relocation counter before attempting to commit an entry to a `Member` state, as a relocation could lead to an incorrect result (relocating the entry already in a `Member` state and thus missed by the uniqueness check) and so the method would need to be restarted as per *Contains* and *Remove*.

Lemma 1: *Contains* is *Linearisable*.

Proof: *Contains* initially loads the relocation count on line **4**, creating a basic snapshot of the bucket. The bucket's bit-mask is loaded on line **7**, and each entry is checked. *Contains* loads the key on line **15**, and is the *linearisation* point for a successful *Contains* call. If a matching key is not found, then the relocation counter is checked again on line **23** to ensure a matching key hasn't been moved around during the search. This re-load indicates whether the snapshot was invalidated during the search, and is the *linearisation* point for an unsuccessful *Contains* call. All code paths in *Contains* have *linearisation* points and thus *Contains* is *linearisable*.

Lemma 2: *Add* is *Linearisable*.

Proof: *Add* is composed of three primary parts and one optional part. The first optional operation can cut *Add* short by determining that a key is already present in the table, and returning `false`. The optional check has the same *linearisation* points at *Contains*. Once a bucket is marked as `Busy` on line **13 - 14**, the algorithm moves the bucket into the appropriate range with repeated calls to *FindCloserBucket*. If *FindCloserBucket* fails to find a bucket, then the table is considered saturated and a resize commences. Once the bucket is within range, the key is written into the bucket and the state is changed to `Inserting`. Once the bucket has transitioned to `Inserting`, then a modified Purcell-Harris exclusivity check is run. There is only one modification point, that being to check the bucket relocation counter before attempting to mark a key as a `Member`. However, this modification doesn't change the *linearisation* point, just whether the method retries. All code paths in *Add* have *linearisation* points and thus *Add* is *linearisable*.

Lemma 3: *Remove* is *Linearisable*.

Proof: *Remove* is similar to *Contains*, except a *CAS* is attempted on the bucket state to move it from `Member` to `Busy`. The *CAS* on line **17** represents the *linearisation* point for a successful remove. *Remove* has the same *linearisation* points as *Contains* when searching for an entry. All code paths in *Remove* have *linearisation* points and thus *Remove* is *linearisable*.

Lemma 4: *FindCloserBucket* is *Linearisable*.

Proof: *FindCloserBucket* attempts to atomically swap the current bucket in a state of `Busy` with another bucket already marked as a `Member`. The method begins by linearly probing from a certain distance away to find a candidate bucket. The bit-mask is checked for possible buckets; buckets which move the bucket being relocated further from its home are excluded (lines **11 - 15**). Once a candidate has been identified, the bucket being relocated is added to the candidate bucket's bit-mask (line **21**) in anticipation of the bucket swap. The swapping of the buckets is executed on line **29** by *K-CAS*. A failed *K-CAS* means that the bucket has either been deleted or moved by a concurrent call, requiring the preemptively set bit to be unset and the method restarted. If the *K-CAS* succeeds, then this is the *linearisation* point of a successful call to *FindCloserBucket*. The method must also remove the old location just moved from the bit-mask, returning the new offsets. If no candidate buckets are found during the linear probe, then the method returns the old offsets. The *linearisation* point for a failed call is the last check of the relocation counters on line **43**. All code paths in *FindCloserBucket* have *linearisation* points and thus *FindCloserBucket* is *linearisable*.

Theorem 1: The hash table is *Linearisable*.

Proof: Each method of the hash table is *linearisable* as per *Lemma 1*, *2*, *3*, and *4*. Hence the hash table is *linearisable*.

4.2 Progress Progress, like correctness, will be argued informally, as the base table of Purcell-Harris already has strong progress arguments accompanying its publication. Both *Contains* and *Remove* methods can be made re-run if run concurrently with a *FindCloserBucket* call relocating an entry from its relevant neighbourhood. This re-run, however, implies the success of another call, thus achieving system progress and lock-freedom. *Remove* tries to *CAS* the state variable into `Busy` from `Member`, potentially restarting if the *CAS* fails. Failure here means the success of another *Remove* or a relocation in *FindCloserBucket*; either way, progress has been achieved. *FindCloserBucket* has the same behaviour as *Contains* and *Remove*, that is, re-running if any relocation counters have been changed since the initial snapshot. *FindCloserBucket* also re-runs if the call to *K-CAS* fails. Such failure only occurs if another method changed the state variable (*Remove* or another *FindCloserBucket*), meaning some other process made progress. The main components of the *Add* method are also lock-free. A simple linear probe with a *CAS* loop will contend with other linear probes, but the failure of one means the success of another, meeting the standards of lock-freedom. Finally, as per standard Purcell-Harris, the uniqueness check is lock-free. Our extra step of checking the relocation counter forces the method to run again, which implies that some other method has made progress on the object, again ensuring lock-freedom. On the whole, we argue that since all methods are lock-free, then the object as a whole must be lock-free.

5 Performance and Discussion

In this section we detail the performance and implementation of our algorithm. All of our code is made freely available online [8]. This includes lock-free Hopscotch Hashing, implementations of alternative competing algorithms (either coded by us or obtained via online sources), and microbenchmarking code which allows readers to replicate our results.

5.1 Experimental Setup For our experiments we opted to use a set of microbenchmarks stressing the hash-table under various capacities and workloads. Our benchmarks were run on a 4 CPU machine, with each CPU (Intel(R) Xeon(R) CPU E7-8890 v3) featuring 18 cores with two hardware threads, and a total of 512 GiB RAM. The machine ran Ubuntu 14.04 with a Linux Kernel version of 3.13.0-141. Each thread was pinned to a specific core for the duration of the test, and threads were scaled in increments of 9, from 9 to 144 threads. When scaling the number of threads, care was taken to pin the thread to an unused core instead of exercising *HyperThreading*^{TM}. We avoided the use of *HyperThreading*^{TM} for as long as possible, scheduling threads to another *NUMA* CPU to avoid its use. *HyperThreading*^{TM} was only used after every core on each CPU had one thread pinned to it *HyperThreading*^{TM}.

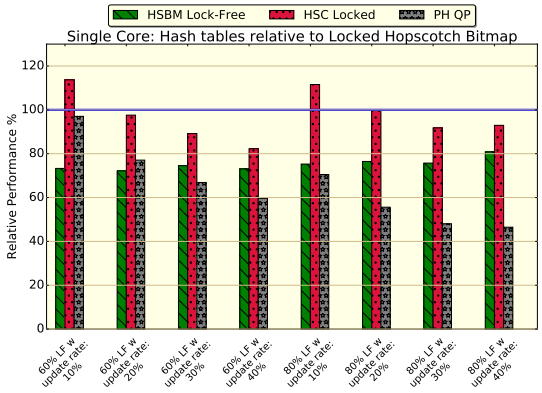
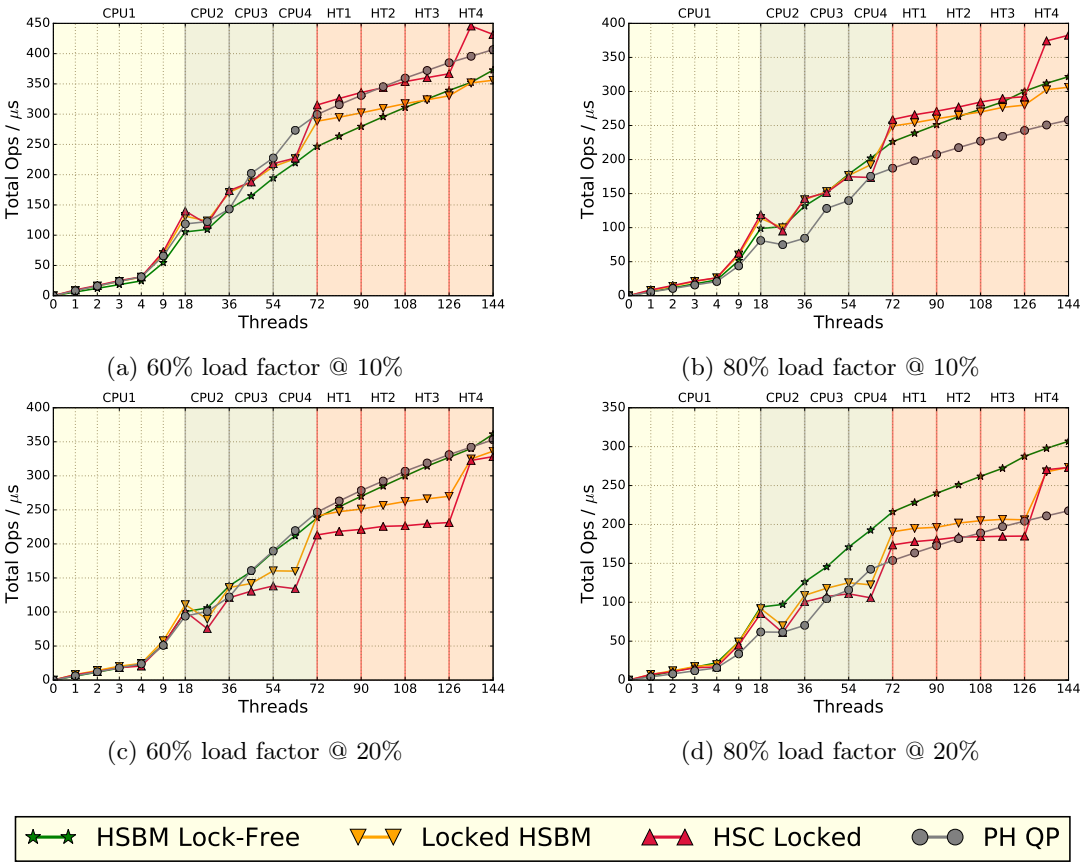


Figure 11: Single thread performance relative to Locked Hopscotch Bit-Map.

The hash-table algorithms benchmarked include both blocking Hopscotch Hashing with fixed-size bit-masks (HSBM Locked), the relative offset variant with probe-chain compression (HSC Locked) [10], the Purcell-Harris Quadratic Probing (PH QP) hash-table [5], and our lock-free Hopscotch Hashing (HSBM Lock-Free).

A number of workload configurations were used in graphing the results. Two load factors of 60% and 80% were chosen, along with four read/write workload configurations, namely 90% reads to 10% updates, 80% reads to 20% updates, 70% reads to 30% updates, and 60% reads to 40% updates. Updates consist of balanced insertions and deletions. All workload configurations were benchmarked at the specified load factors. We sized the tables at 2^{25}, as carried out in [6]. This meant that the table wouldn't fit into the cache, thus highlighting the effective cache use achieved by each algorithm. No memory reclaimer was used, as none was necessary. We used the `numactl` command to mitigate any negative *NUMA* memory effects. This command specified that allocation could only be carried out on the RAM banks closest to the running CPUs once they came into use following increased thread counts.

Concurrent benchmarking has been refined over the years. We strive to perform a like for like comparison, hence our load factors, read/write workloads, and testing process are similar to a number of other previous concurrent hash-table publications [11], [6], [10]. The process was carried out as follows. Each thread



(a) 60% load factor @ 10%

(b) 80% load factor @ 10%

(c) 60% load factor @ 20%

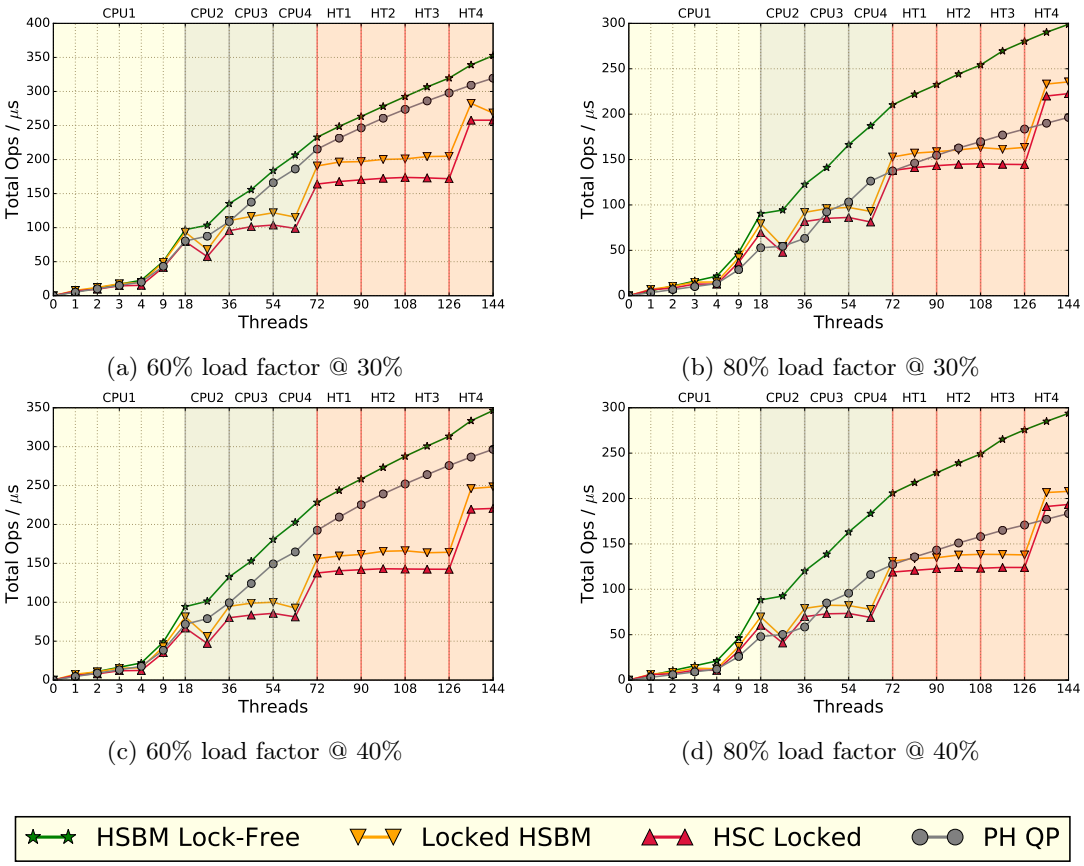
(d) 80% load factor @ 20%

Figure 12: Performance graphs for low update rate.

called a random method with a random argument from some predefined method and key distribution. All threads were synchronised before execution on the data-structure, and executed for a specified amount of time, rather than a specific number of iterations. Each thread counted the number of operations it performed on the structure during the benchmark. The total amount of operations per microsecond for all threads was then graphed, showing throughput. Each experiment was run five times for 10 seconds each, and the average of each result was computed and plotted. All of our algorithms were written in C++11, and compiled with *g++* 4.9.4. The compiler had `O3` level of optimisation, and also targeted the specific processor architecture it was being run on.

5.2 Results The results are listed in Figures **12** and **13**, showing the throughput as a function of concurrent threads. Each graph shows the throughput of each algorithm at a lower thread count of 1 - 4 thread(s), so as to better understand the lower scaling. As the number of threads increases, they cross *NUMA* boundaries at multiples of 18. These boundaries are marked by faint grey lines and a background shading in each figure. Once the number of threads exhausts all physical cores, it begins to use *HyperThreading*^{TM} at 72+ threads. The graphs have another faint red line and background shading to indicate when a new CPU activates *HyperThreading*^{TM}.

We attempt to identify common trends in all graphs before addressing the specifics of each benchmark result. The common pattern is that, at low thread counts, all algorithms are very competitive in terms of performance. The algorithms typically stay close together in performance up until the number of threads scheduled requires the use of another CPU (18 < threads). There are several "kinks" in the graph which are common throughout. All algorithms suffer performance penalties when using an extra CPU (18 < threads, 36 < threads, and 54 < threads). The penalty is either a dip in overall performance or a reduction in the slope of the performance line. As the number of threads increases, all algorithms demonstrate continued slowing in performance, with the angle of the line decreasing fur-



(a) 60% load factor @ 30%

(b) 80% load factor @ 30%

(c) 60% load factor @ 40%

(d) 80% load factor @ 40%

Figure 13: Performance graphs for higher update.

ther when *HyperThreading*^{TM} is engaged (72+ threads). The reader should also note that the throughput scaling in each graph is different. In Figure **11** we highlight the performance of each algorithm using only 1 thread relative to the locked bit-map Hopscotch implementation. The results show that the lock-free algorithms perform significantly worse when using a single thread. This would hint that the majority of "work" being done is for scalability when more threads are scheduled.

Another common trend is that, as load factor increases, the performance of a table decreases. As the table fills up with entries, the cost of each operation also increases. The number of collisions goes up, more entries need to be checked, and generally more work is done. The Purcell-Harris table is hit particularly hard by an increase in load factor. The quadratic probing algorithm needs to check substantially more entries than the equivalent Hopscotch tables, leading to a severe drop in performance as load factor increases. As the update rate climbs, the performances of all tables drop, matching the typical expectation for concurrent objects. The locking Hopscotch tables are hit the hardest from the increase in update ratio, doing best under the lightest updates, with lock-free Hopscotch doing the worst. The reason the lock-free algorithms fare worse at light update rates is that they do more work that allows for greater scalability under heavier load. Lastly, the reader will notice the large jumps in the performance of both locked variants of Hopscotch Hashing. The reason here is that the number of locks is set to the number of active threads, as per the original paper [10]. We, however, apply an optimisation which increases the number to the closet power of two. As a result, thread counts near but just under a power of two suffer a performance drop, while those just after see a large increase: the total amount of concurrency in the hash-table has doubled, while the amount of threads has only increased by 9.

Each performance graph is grouped by the update rate and load factor. Accordingly, we analyse the performance according to that grouping. The following analysis refers to the figures in Figure **12**. A mixed bag of winners is evident, as every table switches out for first place depending on the load factor or update

rate. Generally, at lower update rates locked Hopscotch fares well, as seen in Figures **12a** and **12b**. As the update rate increases, the lock-free algorithms start to pull ahead. Figure **12c** has the lock-free algorithms drawing for first place throughout most of the graph, only to be bested at the last configuration. Figure **12d** shows quadratic probing falling off, with lock-free Hopscotch growing and maintaining its lead over the locked variants in Figure **12d**. We switch focus now to the Figures in **13**. Broadly speaking, at a higher update rate the throughputs of all algorithms are reduced. The locked variants of Hopscotch Hashing have a significant performance drop at all load factors and update rates. Similarly, the performance of Lock-Free Hopscotch Hashing decreases as the load factor goes up. The gap between Purcell-Harris Quadratic Probing and Lock-Free Hopscotch Hashing grows, with lock-free Hopscotch strengthening its lead at 60% and 80% load factor when the update rate increases.

As is apparent from Figures **12** and **13**, lock-free Hopscotch starts slow but ends up dominating in terms of performance. The lock-free Purcell-Harris quadratic-probing has a good showing, but drops significantly in performance at the higher load factors. Although our algorithm is slower than the locked Hopscotch and Purcell-Harris tables at the lower updates rates, as both the load factor and update rate increase, our algorithm pulls ahead of the competition. Locked Hopscotch performs best at low update rates and is very strong at all load factors. It performs consistently at each load factor, though suffers considerably under heavy update rates. Overall, the lock-free Hopscotch solution finishes either roughly 20% behind locked Hopscotch, or 50% ahead at the highest thread count.

6 Conclusion and Future Work

We have presented a lock-free Hopscotch Hashing algorithm which achieves noteworthy performance relative to other lock-free and concurrent algorithms. To the best of our knowledge, this is the first presentation of such an algorithm in the literature. Our experiments show that the approach is competitive with locked Hopscotch at low updates, and dominates above that. The algorithm is relatively simple and just as portable as competitors, needing only single word compare-and-swap instructions. In future work we plan to create a lock-free relative-offset variant and larger bit-mask to potentially improve performance.

Acknowledgements.

We wish to thank Nir Shavit for his advice and use of his computing cluster when generating results. We also wish to thank William M. Leiserson for his feedback and critique of this work, as well as his friendship and support. Finally we would like to thank the Irish Centre for High-End Computing (ICHEC) for the use of their machines for benchmarking.

References

[1] Harris, T. and Fraser, K. and Pratt, I., *A Practical Multi-word Compare-and-Swap Operation*, Proceedings of the 16th International Conference on Distributed Computing, 2002, pp. 265–279

[2] Gao, Hui and Groote, Jan and Hesselink, Wim., *Lock-free dynamic hash tables with open addressing*, Distributed Computing, 2003, pp. 21–42

[3] Shalev, O. and Shavit, N., *Split-Ordered Lists: Lock-Free Extensible Hash Tables*, Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing, 2003

[4] Feldmanr S., Pierre L., and Damian D., *Concurrent multi-level arrays: Wait-free extensible hash maps*, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2013

[5] Purcell, C. and Harris, T., *Non-blocking Hashtables with Open Addressing*, Distributed Computing: 19th International Conference, DISC, September 2005, pp. 26–29

[6] Nielsen, J. and Karlsson, S., *A Scalable Lock-free Hash Table with Open Addressing*, SIGPLAN Not., 2016, pp. 33:1–33:2

[7] Herlihy, M. and Wing, J., *Linearizability: A Correctness Condition for Concurrent Objects*, ACM Trans. Program. Lang. Syst., 1990, pp. 463–492

[8] Kelly R., *Source Code For Lock-Free Hopscotch Hashing Benchmark*, `https://github.com/DaKellyFella/LockFreeHopscotchHashing`

[9] Click C., *Lock-Free / Wait-Free Hash Table*, `http://web.stanford.edu/class/ee380/Abstracts/070221_LockFreeHash.pdf`

[10] Herlihy, M. and Shavit, N. and Tzafrir, M., *Hopscotch Hashing*, Proceedings of the 22nd international symposium on Distributed Computing, 2008, pp. 350–364

[11] Robert Kelly and Barak A. Pearlmutter and Phil Maguire, *Concurrent Robin Hood Hashing*, 22nd International Conference on Principles of Distributed Systems, 2018, pp. 10:1–10:16

[12] Nguyen, Nhan and Tsigas, Philippas, *Lock-Free Cuckoo Hashing*, Proceedings of the IEEE 34th International Conference on Distributed Computing Systems, 2014, pp. 627–636

[13] Michael, M., *High performance dynamic lock-free hash tables and list-based sets*, Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, 2002

[14] Knuth, D., *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*, Addison Wesley Longman Publishing Co., Inc., 1997

[15] Cormen, T. and Stein, C. and Rivest, R. and Leiserson, C., *Introduction to Algorithms*, McGraw-Hill Higher Education, 2001
[16] Padegs, A., *System/370 extended architecture: Design considerations*, IBM Journal of Research and Development, 1983, pp. 192–205
[17] Herlihy, M. and Luchangco, V. and Moir, M., *Obstruction-Free Synchronization: Double-Ended Queues As an Example*, Proceedings of the 23rd International Conference on Distributed Computing Systems, 2003
[18] Herlihy, M. and Shavit, N., *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008
[19] Alistarh, D. and Leiserson, W. and Matveev, A. and Shavit, N., *Forkscan: Conservative Memory Reclamation for Modern Operating Systems*, EuroSys, 2017
[20] Alistarh, D. and Leiserson, W. and Matveev, A. and Shavit, N., *ThreadScan: Automatic and Scalable Memory Reclamation*, SPAA, 2015
[21] Leino, K. Rustan M. and Müller, Peter, *A Basis for Verifying Multi-threaded Programs*, Programming Languages and Systems: 18th European Symposium on Programming, 2009, pp. 378–393
[22] A. Amighi and S. Blom and M. Huisman, *VerCors: A Layered Approach to Practical Verification of Concurrent Software*, 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, 2016
[23] Cohen, N. and Petrank, E., *Efficient memory management for lock-free data structures with optimistic access*, Proceedings of the 27th ACM symposium on Parallelism in Algorithms and Architectures, 2015, pp. 254–263
[24] Fraser, K., *Practical lock-freedom*, University of Cambridge, Computer Laboratory, 2004
[25] Michael, M., *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, IEEE Trans. Parallel Distrib. Syst., 2004, pp. 491–504
[26] M. Arbel-Raviv and T. Brown, *Reuse, Don't Recycle: Transforming Lock-Free Algorithms That Throw Away Descriptors*, DISC, 2017
[27] Herlihy, M., *Wait-free synchronization*, ACM Transactions on Programming Languages and Systems (TOPLAS), 13(1), 124-149, 1991.
[28] Peterson, W. W. (1957). *Addressing for random-access storage* IBM journal of Research and Development, 1(2), 130-146, 1957
[29] P., Rasmus and R. Flemming Friche *Cuckoo Hashing*, J. Algorithms, 2004

A Appendix

A.1 Additional performance results Our testing was performed on another machine to ensure we weren't fitting to a particular hardware architecture during our benchmarking. The machine had 2 CPUs (Intel® Xeon® Gold 6148) with 20 cores each, and 27.5MB of L3 Cache. Each core had two hardware threads, meaning the total number of threads was 80. The machine had 192 GiB of RAM installed. Threads were pinned exactly like our main experiments, except in increments of 5, from 5 to 80. All of our algorithms were compiled with *clang++* 7.0 at `O3` level of optimisation, and also targeted the specific processor architecture they were being run on. Everything else about the testing process remained the same. The figures for single threaded performance can be seen in Figure **14** while the results for throughput as a function of concurrency can be seen in Figure **15**. The results in Figure **15** broadly match the same trends as seen in our results above in Figures **12** and **13**.

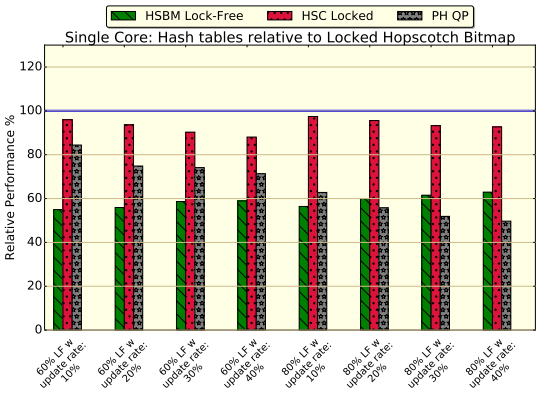
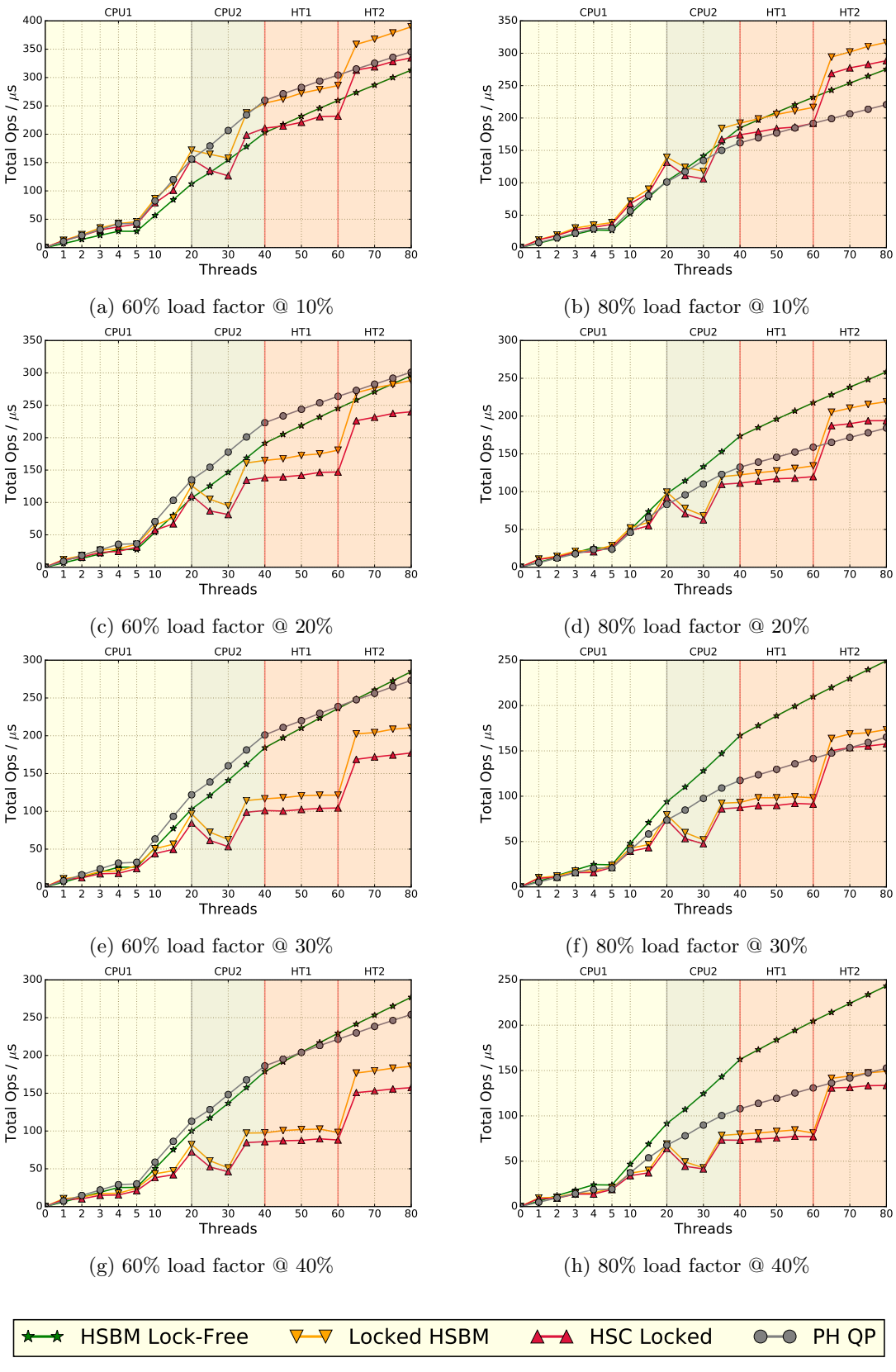


Figure 14: Single thread performance relative to Locked Hopscotch Bit-Map for the second machine.



(a) 60% load factor @ 10%

(b) 80% load factor @ 10%

(c) 60% load factor @ 20%

(d) 80% load factor @ 20%

(e) 60% load factor @ 30%

(f) 80% load factor @ 30%

(g) 60% load factor @ 40%

(h) 80% load factor @ 40%

Figure 15: Performance graphs for the second machine.